universidade de aveiro

# NCD ID: Identifying Songs through Compression Algorithms

Teoria Algorítmica da Informação - 40752

Gonçalo Marques - 98648
João Reis - 115513
Renan Ferreira - 93168

Aveiro
May, 2024

# Table of Contents

# Abstract

# Introduction

In this paper, an application of the Normalised Compression Distance (NCD) will be explored to identify music based on small audio segments. NCD is a computable approximation of the non-computable Normalised Information Distance (NID) through the use of compression algorithms, making it possible to implement and test in real-world scenarios.

Standard compression algorithms are used to determine the efficiency and accuracy of NCD when trying to identify music based on short audio samples, similar to the popular application, Shazam. To do this, the NCD needs to be calculated between a query segment of audio and each song in a database, identifying which song produced the smallest value. This project applies the concepts of Kolmogorov complexity to measure how similar two strings are. This is done using a calculation similar to the formula for NID, given as

$$NID(x, y) = \frac{max\{ K(x|y), K(x|y)\}}{max\{ K(x), K(y)\}},$$

where $K(x)$ is the Kolmogorov complexity of a string $x$ and $K(x|y)$ is the Kolmogorov complexity of x when y is given as context. (Citation from lecturers) Due to the non-computability of the Kolmogorov complexity, the NCD uses a similar formula with compression, presented as

$$NCD(x, y) = \frac{C(x,y) - min\{ C(x), C(y)\}}{max\{ C(x), C(y)\}}.$$

Here, $C(x)$ represents the number of bits needed by a compressor, $C$, to represent a string, $x$ while, $C(x, y)$ is the number of bits needed to compress $x$ and $y$ together. The closer the calculated distances are to zero, the more similar they are.

The set goals for this study are:

- Create a database with at least 25 different songs.
- Use multiple standard compressors to compute NCD values.
- Test the identification system with small audio segments of different sizes.
- Add different levels of noise and test them again.
- Convert audio into more suitable representations for compression.
- Compile and analyse the results.

# Methodology

This project involved identifying songs based on small segments of said songs. To begin this process, however, a database with songs was required. First, a couple of songs were acquired from the Free Music Archive (FMA) for testing purposes. The problem here was that the files were in the *.mp3* format and the programs only worked with *.wav* files. To solve this, a small Python script was created that would convert all files in a folder from *.mp3* to *.wav* and delete the original files. Later, a Python script was created to download music from YouTube through video links to get more music files.

## Segment Extraction

Segments need to be extracted from the original music files to test the efficacy of the NCD calculation for identifying music. So, a small program was developed using pre-existing libraries that would serve this exact purpose.

It starts by parsing the command-line arguments for the input file, the output file name, the starting point in the song, the duration of the segment and the noise level intended (as the default is 0). After this, the script will validate the input file and the segment range and read the specified segment from the file. If opted for, it will add noise to the segment extracted and to finalise it will save this file to the output provided.

```
SegmentExtractor -i <input_file> -o <output_file> -start <start_point>
                 -dur <duration> -n <noise_level>";
```

*Fig 1. Example code to run the Segment Extraction program*

## Music Identification

As previously mentioned, the music here is identified by checking which file is closest to the segment under analysis by calculating the NCD. To do this, compression algorithms are required which, in this study, gzip, bzip2, lzma and zstd were used.
The identification process begins by parsing the command-line arguments for a query file, database directory, compression method, window size, shift, downsampling, and number of frequencies. The last four arguments were kept at their default value for the sake of clarity with the results.

```
MusicIdentifier -q <query_file> -d <database_dir>
                [-m <compression_method>] -ws <window_size>
                 -sh <shift> -ds <downsampling> -nf <num_freqs>
```

*Fig 2. Example code to run the Music Identification program*

After parsing the arguments, the program validates the existence of the query file and the database directory, loads the query audio file and computes its Fast Fourier Transform (FFT) signature. This is a representation of the audio data in the frequency domain, in other words, a binary representation of the most notable frequencies within a specified range.

With the FFT signature of the segment in question, an iterative process begins to compute the FFT signature of the full songs, proceeding to then calculate the NCD and comparing the results, verifying which combination of segment and song produces the smallest value.

# Results

For the purpose of testing the use of NCD calculations for music identification, two random songs were chosen from the database and segments were extracted from them. At random, it was picked for the segments to start at 50 seconds into the songs and, these would either last 1,5 or 10 seconds to track efficacy. More versions of these segments were created as well with added noise. Noise levels were modified to be 0, 100, 1000, 10000.
The following tables show the results obtained results for each compression method.

Table 1, is showing the results using the gzip compressor, it proves itself to be highly inaccurate for both songs other than being oddly precise for the 1-second duration up until a certain threshold of noise. However, a pattern will soon emerge when looking at the other results.

| Compressor | Duration | noise | Hit (song 1) | Hit (song 2) |
|---|---|---|---|---|
| gzip | 1 | 0 | No | Yes |
| | | 100 | No | Yes |
| | | 1000 | No | Yes |
| | | 10000 | No | No |
| | 5 | 0 | No | No |
| | | 100 | No | No |
| | | 1000 | No | No |
| | | 10000 | No | No |
| | 10 | 0 | No | No |
| | | 100 | No | No |
| | | 1000 | No | No |
| | | 10000 | No | No |

*Table 1. GZIP compressor results*

Now, having a look at Table 2, it is possible to tell that even though there are some correct guesses, the model was generally imprecise and severely inefficient. Once again, slightly precise with a very minute duration on segments of song 2.

| Compressor | Duration | noise | Hit (song 1) | Hit (song 2) |
|---|---|---|---|---|
| bzip2 | 1 | 0 | No | Yes |
| | | 100 | No | Yes |
| | | 1000 | No | No |
| | | 10000 | No | No |
| | 5 | 0 | No | No |
| | | 100 | No | No |
| | | 1000 | No | Yes |
| | | 10000 | No | No |
| | 10 | 0 | No | No |
| | | 100 | No | No |
| | | 1000 | No | No |
| | | 10000 | No | No |

*Table 2. BZIP2 compressor results*

Both Tables 3 and 4, using LZMA and ZSTD, respectively, provided similarly curious results. This time, the accuracy was more focussed on song 1 but, no real conclusion can be taken from them.

| Compressor | Duration | noise | Hit (song 1) | Hit (song 2) |
|---|---|---|---|---|
| lzma | 1 | 0 | No | No |
| | | 100 | No | No |
| | | 1000 | No | No |
| | | 10000 | No | No |
| | 5 | 0 | No | No |
| | | 100 | Yes | No |
| | | 1000 | No | No |
| | | 10000 | Yes | No |
| | 10 | 0 | No | No |
| | | 100 | No | No |
| | | 1000 | No | No |
| | | 10000 | No | No |

*Table 3. LZMA compressor results*

| Compressor | Duration | noise | Hit (song 1) | Hit (song 2) |
|---|---|---|---|---|
| zstd | 1 | 0 | No | No |
| | | 100 | No | No |
| | | 1000 | No | No |
| | | 10000 | No | No |
| | 5 | 0 | No | No |
| | | 100 | No | No |
| | | 1000 | No | No |
| | | 10000 | No | No |
| | 10 | 0 | Yes | No |
| | | 100 | Yes | No |
| | | 1000 | Yes | No |
| | | 10000 | No | No |

*Table 4. ZSTD compressor results*

# Discussion

Upon looking at the results, they reveal significant inaccuracies, which are particularly evident. The NCD values, which were expected to be fairly low for correct guesses, were actually above 0.95. This either shows poor performance by the program or that the segments are too small. In either case, it indicates that the current NCD implementation is not good for audio identification.

Multiple factors could contribute to the high NCD values and lack of precision in the guesses. One potential problem is the parameter settings used for computing the FFT signatures. Parameters such as window size, shift, downsampling, and the number of frequencies can significantly impact the FFT analysis and, incorrect settings may lead to less-than-optimal feature extraction and therefore, affect accuracy. Changing the values, namely by increasing shift a lot,  made it possible to obtain much smaller NCD values. However, this only hurt the accuracy by making it much worse.

Relying simply on FFT signatures may not be sufficient for a robust and reliable audio identification system. The FFT signatures provide a representation of the signal's frequency but do not present any other important audio characteristics. Additional audio features could enhance the performance of the identification process.

For example, Mel-frequency cepstral coefficients (MFCCs) are used in speech and audio processing due to the ability to capture the timbral texture of sounds. (Mermelstein, 1976) Similarly, chrome features can represent the twelve different pitch classes and are good at capturing the harmonic and melodic content in music. (Ellis, 2007) Through the addition of these features in the feature extraction process, the system could, perhaps, gain a better understanding of the audio signals, leading to improved identification precision.

# Conclusions

In conclusion, the current implementation of the program, based on NCD calculations through the use of FFT signatures and standard compression algorithms, has proved itself limited in its effectiveness. By only producing large numbers of NCD it suggests poor performance and that, as it is currently applied, may not be suited or ideal as a metric for audio identification.

To possibly improve the performance, optimisation of the FFT parameter settings could be necessary. On top of that, integrating advanced audio features such as MFCCs and chroma features could give the algorithm and more detailed representation of the audio signal, thus significantly improving its accuracy.

# Future Work

For future reference, experimentation with these new additional features should be the focus, as well as further optimising the parameter settings in order to get a more reliable music identifier. Using more advanced and diverse audio features may help to overcome the currently observed limitations and, in return, get better, more accurate results.

# References


Ellis, Daniel P. W. “Classifying music audio with timbral and chroma features.” *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 2007, pp. 339-340, https://www.ee.columbia.edu/~dpwe/pubs/Ellis07-timbrechroma.pdf.

FMA. *Free Music Archive: Royalty free music and 'free to download' music*, Tribe of Noise, http://freemusicarchive.org.

Mermelstein, P. “Distance measures for speech recognition, psychological and instrumental.” *Pattern Recognition and Artificial Intelligence*, 1976, pp. 374-388, https://cir.nii.ac.jp/crid/1570291226078527872.

# Appendix

Script to download music from YouTube:

```
from pytube import YouTube
from moviepy.editor import *
import os

def download_audio(url, output_path):
    try:
        yt = YouTube(url)
        # Get the highest resolution audio stream
        audio_stream = yt.streams.filter(only_audio=True).order_by('abr').desc().first()
        # Download the audio
        audio_file = audio_stream.download(output_path=output_path)
        # Convert the audio to mp3
        audio_clip = AudioFileClip(audio_file)
        mp3_file = os.path.join(output_path, yt.title + ".mp3")
        audio_clip.write_audiofile(mp3_file)
        audio_clip.close()
        print("Audio downloaded successfully as", yt.title + ".mp3")
        delete_original(audio_file)

    except Exception as e:
        delete_original(audio_file)
        print("Error:", str(e))

def delete_original(audio_file):
    # Delete the original audio file
        os.remove(audio_file)
        print("Original audio file deleted successfully")

if __name__ == "__main__":
    url = input("Enter the YouTube video URL: ")
    output_path = "C:\\Users\\jprw3\\Downloads\\Music"
    download_audio(url, output_path)
```

Script to convert MP3 files into WAV:

```python
import os
import soundfile as sf
import librosa

def convert_mp3_to_wav(folder_path):
    for filename in os.listdir(folder_path):
        if filename.endswith('.mp3'):
            mp3_path = os.path.join(folder_path, filename)
            wav_path = os.path.join(folder_path, filename[:-4] + '.wav')

            # Load the mp3 file
            data, samplerate = librosa.load(mp3_path, sr=None)

            # Save as wav file
            sf.write(wav_path, data, samplerate)

            # Delete the original mp3 file
            os.remove(mp3_path)
            print(f"Converted and deleted: {filename}")

if __name__ == "__main__":
    folder_path = input("Enter the path to the folder containing MP3 files: ")
    convert_mp3_to_wav(folder_path)
```